APPLICATION

FOR

UNITED STATES LETTERS PATENT

TITLE: CONTEXT PIPELINES

APPLICANT: GILBERT WOLRICH, MARK B. ROSENBLUTH, MATTHEW J. ADILETTA, DEBRA BERNSTEIN AND HUGH M. WILKINSON III

CERTIFICATE OF MAILING BY EXPRESS MAIL

Express Mail Label No. EL870691432US

I hereby certify that this correspondence is being deposited with the United States Postal Service as Express Mail Post Office to Addressee with sufficient postage on the date indicated below and is addressed to the Commissioner for Patents, Washington, D.C. 20231.

January 25, 2002
Date of Deposit

Signature

Gabriel Lewis
Typed or Printed Name of Person Signing Certificate

CONTEXT PIPELINES

BACKGROUND

Parallel processing is an efficient form of information processing of concurrent events in a computing process. Parallel processing demands concurrent execution of many programs in a computer, in contrast to sequential processing. In the context of a parallel processor, parallelism involves doing more than one thing at the same time. Unlike a serial paradigm where all tasks are performed sequentially at a single station or a pipelined machine where tasks are performed at specialized stations, with parallel processing, many stations are provided, each capable of performing various tasks simultaneously. A number of stations work simultaneously and independently on the same or common elements of a computing task. Accordingly, using or applying parallel processing can solve computing tasks.

BRIEF DESCRIPTION OF THE DRAWINGS

FIG. 1 is a block diagram of a communication system employing a hardware-based multithreaded processor.

FIG. 2 is a block diagram of a programming engine functional unit employed in the hardware-based multithreaded processor of FIG. 1.

FIG. 3 is a block diagram of a context state transition.

FIG. 4 is a block diagram of a context addressable memory (CAM).

FIG. 5 is a detailed block diagram of a CAM process.

FIG. 6 is a simplified block diagram of a context pipeline process.

DESCRIPTION

Architecture

Referring to FIG. 1, a computer processing system 10 includes a parallel, hardware-based multithreaded network processor 12. The hardware-based multithreaded processor 12 is coupled to a memory system or memory resource 14. Memory system 14 includes dynamic random access memory (DRAM) 14a and static random access memory 14b (SRAM). The processing system 10 is especially useful for tasks that can be broken into parallel subtasks or functions. Specifically, the hardware-based multithreaded processor 12 is useful for tasks that are bandwidth oriented rather than latency oriented. The hardware-based multithreaded processor 12 has multiple functional microengines or programming engines 16 each with multiple hardware controlled threads that are simultaneously active and independently work on a specific task.

The programming engines 16 each maintain program counters in hardware and states associated with the program counters. Effectively, corresponding sets of context or threads can be simultaneously active on each of the programming engines 16 while only one is actually operating at any one time.

In this example, eight programming engines 16a-16h are illustrated in FIG. 1. Each engine from the programming engines 16a-16h processes eight hardware threads or contexts. The eight programming engines 16a-16h operate with shared resources including memory resource 14 and bus interfaces (not shown). The hardware-based multithreaded processor 12 includes a dynamic random access memory (DRAM) controller 18a and a static random access memory (SRAM) controller 18b. The DRAM memory 14a and DRAM controller 18a are typically used for processing large volumes of data, e.g., processing of network payloads from network packets. The SRAM memory 14b and SRAM controller 18b are used in a networking implementation for low latency, fast access tasks, e.g., accessing look-up tables, memory for the core processor 20, and the like.

The eight programming engines 16a-16h access either the DRAM memory 14a or SRAM memory 14b based on characteristics of the data. Thus, low latency, low bandwidth data is stored in and fetched from SRAM memory 14b, whereas higher bandwidth data for

which latency is not as important, is stored in and fetched from DRAM memory 14a. The programming engines 16a-16h can execute memory reference instructions to either the DRAM controller 18a or SRAM controller 18b.

The hardware-based multithreaded processor 12 also includes a processor core 20 for loading microcode control for the programming engines 16a-16h. In this example, the processor core 20 is an XScale™ based architecture.

The processor core 20 performs general purpose computer type functions such as handling protocols, exceptions, and extra support for packet processing where the programming engines 16 pass the packets off for more detailed processing such as in boundary conditions.

The processor core 20 has an operating system (not shown). Through the operating system (OS), the processor core 20 can call functions to operate on the programming engines 16a-16h. The processor core 20 can use any supported OS, in particular, a real time OS. For the core processor 20 implemented as an XScale™ architecture, operating systems such as Microsoft NT real-time, VXWorks and µCOS, or a freeware OS available over the Internet can be used.

Advantages of hardware multithreading can be explained by SRAM or DRAM memory accesses. As an example, an SRAM access

requested by a context (e.g., Thread_0), from one of the programming engines 16 will cause the SRAM controller 18b to initiate an access to the SRAM memory 14b. The SRAM controller 18b accesses the SRAM memory 14b, fetches the data from the SRAM memory 14b, and returns data to a requesting programming engine 16.

During an SRAM access, if one of the programming engines 16a-16h had only a single thread that could operate, that programming engine would be dormant until data was returned from the SRAM memory 14b.

By employing hardware context swapping within each of the programming engines 16a-16h, the hardware context swapping enables other contexts with unique program counters to execute in that same programming engine. Thus, another thread e.g., Thread_1 can function while the first thread, Thread_0, is awaiting the read data to return. During execution, Thread_1 may access the DRAM memory 14a. While Thread_1 operates on the DRAM unit, and Thread_0 is operating on the SRAM unit, a new thread, e.g., Thread_2 can now operate in the programming engine 16. Thread_2 can operate for a certain amount of time until it needs to access memory or perform some other long latency operation, such as making an access to a bus interface. Therefore, simultaneously, the multi-threaded processor 12 can have a bus

operation, an SRAM operation, and a DRAM operation all being completed or operated upon by one of the programming engines 16 and have one more threads or contexts available to process more work.

The hardware context swapping also synchronizes the completion of tasks. For example, two threads can access the shared memory resource, e.g., the SRAM memory 14b. Each one of the separate functional units, e.g., the SRAM controller 18b, and the DRAM controller 18a, when they complete a requested task from one of the programming engine threads or contexts reports back a flag signaling completion of an operation. When the programming engines 16a-16h receive the flag, the programming engines 16a-16h can determine which thread to turn on.

One example of an application for the hardware-based multithreaded processor 12 is as a network processor. As a network processor, the hardware-based multithreaded processor 12 interfaces to network devices such as a Media Access Controller (MAC) device, e.g., a 10/100BaseT Octal MAC 13a or a Gigabit Ethernet device 13b. In general, as a network processor, the hardware-based multithreaded processor 12 can interface to any type of communication device or interface that receives or sends large amount of data. The computer processing system 10 functioning in a networking application can receive network

packets and process those packets in a parallel manner.

Programming Engines

Referring to FIG. 2, one exemplary programming engine 16a from the programming engines 16a-16h, is shown. The programming engine 16a includes a control store 30, which in one example includes a RAM of 4096 instructions, each of which is 40-bits wide. The RAM stores a microprogram that the programming engine 16a executes. The microprogram in the control store 30 is loadable by the processor core 20 (FIG. 1).

In addition to event signals that are local to an executing thread, the programming engine 16a employs signaling states that are global. With signaling states, an executing thread can broadcast a signal state to all programming engines 16a-16h. Any and all threads in the programming engines can branch on these signaling states. These signaling states can be used to determine availability of a resource or whether a resource is due for servicing. The context event logic has arbitration for the eight (8) threads. In one example, the arbitration is a round robin mechanism. Other techniques could be used including priority queuing or weighted fair queuing.

As described above, the programming engine 16a supports multi-threaded execution of eight contexts. This allows one

thread to start executing just after another thread issues a memory reference and must wait until that reference completes before doing more work. Multi-threaded execution is critical to maintaining efficient hardware execution of the programming engine 16a because memory latency is significant. Multi-threaded execution allows the programming engines 16 to hide memory latency by performing useful independent work across several threads.

The programming engine 16a, to allow for efficient context swapping, has its own register set, program counter, and context specific local registers. Having a copy per context eliminates the need to move context specific information to and from shared memory and programming engine registers for each context swap. Fast context swapping allows a context to do computation while other contexts wait for input-output (I/O), typically, external memory accesses to complete or for a signal from another context or hardware unit.

For example, the programming engine 16a executes the eight contexts by maintaining eight program counters and eight context relative sets of registers. There can be six different types of context relative registers, namely, general purpose registers (GPRs) 32, inter-programming agent registers (not shown), Static Random Access Memory (SRAM) input transfer registers 34, Dynamic

Random Access Memory (DRAM) input transfer registers 36, SRAM output transfer registers 38, DRAM output transfer registers 40.

The GPRs 32 are used for general programming purposes. The GPRs 32 are read and written exclusively under program control. The GPRs 32, when used as a source in an instruction, supply operands to an execution datapath 44. When used as a destination in an instruction, the GPRs 32 are written with the result of the execution datapath 44. The programming engine 16a also includes I/O transfer registers 34, 36, 38 and 40 which are used for transferring data to and from the programming engine 16a and locations external to the programming engines 16a, e.g., the DRAM memory 14a, the SRAM memory 14b, and etc.

A local memory 42 is also used. The local memory 42 is addressable storage located in the programming engine 16a. The local memory 42 is read and written exclusively under program control. The local memory 42 also includes variables shared by all the programming engines 16a-16h. Shared variables are modified in various assigned tasks during functional pipeline stages by the programming engines 16a-16h, which are described next. The shared variables include a critical section, defining the read-modify-write times. The implementation and use of the critical section in the computing processing system 10 is also described below.

Programming Engine Contexts

Each of the programming engine 16 supports multi-threaded execution of eight contexts. One reason for this is to allow one thread to start executing just after another thread issues a memory reference and must wait until that reference completes before doing more work. This behavior is critical to maintaining efficient hardware execution of the programming engines 16a-16f because memory latency is significant. Stated differently, if only a single thread execution was supported, the programming engine would sit idle for a significant number of cycles waiting for references to complete and thereby reduce overall computational throughput. Multi-threaded execution allows a programming engine to hide memory latency by performing useful independent work across several threads.

The programming engines 16a-16h each have eight available contexts. To allow for efficient context swapping, each of the eight contexts in the programming engine has its own register set, program counter, and context specific local registers. Having a copy per context eliminates the need to move context specific information to/from shared memory and programming engine registers for each context swap. Fast context swapping allows a context to do computation while other contexts wait for I/O,

typically external memory accesses, to complete or for a signal from another context or hardware unit.

Accordingly, FIG. 3 illustrates the state transitions 100 for a context. Each of the eight contexts will be in one of the states described above. At most, one context can be in the executing state at a time and any number of contexts can be in any of the other states:

1) Inactive state (100a) – Because some applications may not require all eight contexts, a context is in the inactive state when its CTX_Enable (Context Enable) Control and Status Register (CSR) enable bit is a '0'.

2) Ready state (100b) – In this state, although a context is ready to execute, the context cannot proceed because a different context is still executing. When the executing context goes to a sleep state, the programming engine's context arbiter selects the next context to go to the executing state from among all the contexts in the ready state. The arbitration is round robin.

3) Executing state (100c) – A context is in an executing state when its context number is in Active_CTX_Status CSR. The executing context's programming counter (not shown) is used to fetch instructions from the control store 50. A context remains in the executing state until it executes an instruction that causes it to enter the sleep state. At most, one context can be

in the executing state at any time.

4) Sleep state (100d) – A context is waiting for external event(s) specified in the CTX_#_Wakeup_Events CSR to occur where # indicates eight different contexts such as context #0 through #7, typically, but not limited to, an I/O access. In this state, the context does not arbitrate to enter the executing state.

Returning to FIG. 2, each programming engine 22 includes four types of 32-bit datapath registers as described below. The 256 general purpose registers, 128 next neighbor registers, 512 transfer registers, and 640 32-bit words of local memory.

General Purpose Registers

The programming engine 16a includes General Purpose Registers (GPRs) 52 which are used for general programming purposes. They are read and written exclusively under program control. The GPRs 52, when used as a source in an instruction, supply operands to an execution datapath 56. When used as a destination in an instruction, the GPRs 52 are written with the result of the execution datapath 56. The GPRs 52 are physically and logically contained in two banks, GPR A 52a and GPR B 52b, as illustrated in FIG. 3.

Transfer Registers

The programming engine 16a also includes transfer registers 58 and 60. Transfer registers 34, 36, 38 and 40 are used for transferring data to and from the programming engine 16a and locations external to the programming engine, e.g., DRAMs, SRAMs etc. There are four types of transfer registers as illustrated in FIG. 2, namely, input transfer registers and output transfer registers.

The input transfer registers when used as a source in an instruction, supply operands to the execution datapath 44. The output transfer registers when used as a destination in an instruction, are written with the result from the execution datapath 44.

Local Control and Status Registers (CSRs)

Local control and status registers (CSRs) 66 are external to the execution data path 56 and hold specific purpose information. They can be read and written by special instructions (local_csr_rd and local_csr_wr) and are typically accessed less frequently than datapath registers.

Next Neighbor Registers

The programming engine 16a also includes 128 Next Neighbor (NN) registers 54. Each NN Register 54, when used as a source in

an instruction, also supplies operands to the execution datapath 44. The NN register 54 is written either by an external entity, not limited to, an adjacent programming engine, or by the same programming engine 16a where the NN register 54 resides. The specific register is selected by a context-relative operation where the register number is encoded in the instruction, or as a ring operation, selected via, e.g., NN_Put (NN write address) and NN_Get (NN read address) in the CSR Registers.

NN_Put registers are used when the previous neighboring programming engine executes an instruction with NN_Put as a destination. The NN register 54 selected by the value in this register is written, and the value in NN_Put is then incremented (a value of 127 wraps back to 0). The value in this register is compared to the value in NN_Get register to determine when to assert NN_Full and NN_Empty status signals.

NN_Get registers are used when the NN register 54 is accessed as a source, which is specified in the source field of the instruction. The NN register 54 selected by the value in this register is read, and the value in NN_Put is then incremented (a value of 127 wraps back to 0). The value in this register is compared to the value in the NN_Put register to determine when to assert NN_Full and NN_Empty status signals.

Specifically, when the NN register 54 is used as a

destination in an instruction, the instruction result data is sent out of the programming engine 16a, typically to another, adjacent programming engine. On the other hand, when the NN register 54 is used as a destination in an instruction, the instruction result data is written to the selected NN Register 54 in the programming engine 16a. The data is not sent out of the programming engine 22f as it would be when the NN register 54 is used as a destination. The NN register 54 is used in a context pipelining method, as described below.

Local Memory

The programming engine 16a includes 640 32-bit words in a local memory 42. Local memory 42 is addressable storage located in the programming engine 16a. The local memory 42 is read and written exclusively under program control. The local memory 42 supplies operands to the execution datapath 44 as a source, and receives results as a destination. The specific local memory location selected is based on the value in one of the LM_Addr registers 53, which is written by local_CSR_wr instructions. There are two LM_Addr registers 53 per context and a working copy of each. When a context goes to the sleep state 100d, the value of the working copies is put into the context's copy of LM_Addr. When the context goes to the executing state, the value in its

copy of LM_Addr is put into the working copies. The choice of LM_Addr_0 or LM_Addr_1 is selected in the instruction. It is also possible to make use of both or one LM_Addrs as global by setting CTX_Enable [LM_Addr_0_Global] and/or CTX_Enable [LM_Addr_1_Global]. When used globally, all contexts use the working copy of LM_Addr in place of their own context specific copy.

The local memory 42 also includes variables shared by the programming engines 16a-16h. Shared variables are modified in various assigned tasks used during pipeline stages by the programming engines 16a-16h. The shared variables include a critical section which defines their read-modify-write times.

Critical Section

The pipeline stages of the programming engines 16a-16h include a minimum resolution defined by the difference between the critical section of the shared variables and the arrival time of a subsequent packet. The time allotted to the critical section must be less than the arrival time of the subsequent packet, which determines the minimum resolution of the pipeline stage. The latency of a memory read followed by the instructions to modify a variable, followed by a write, far exceeds the packet arrival rate for minimum size packets. Therefore, the critical

section must be maintained to be less than the arrival time of the subsequent packet.

Execution Data Path

The programming engine 16a also includes the execution data path 44 that can take one or two operands, perform an operation, and optionally write back a result. The sources and destinations can be GPRs 32, transfer registers 34, 36, 38, and 40, NN register 54, and the local memory 42. The operations are shifts, add/subtract, logicals, multiply, byte align, and find first one bit. The execution data path 44 also includes a content addressable memory (CAM) 64.

Context Addressable Memory (CAM)

Fig. 4 illustrates a CAM block diagram 102. The programming engine 16a includes the 16 entry CAM 64 with associated control logic 104. Each entry stores a 32-bit value, which can be compared against a source operand. All entries are compared in parallel and the result of the lookup is a 6-bit value. The 6-bit result consists of a 2-bit code concatenated with 4-bit entry number 106. Possible results of a lookup 108 are two fold. A first result is a miss (0) 110 where the lookup value is not in the CAM 64 and the entry number is the Least Recently Used (LRU)

entry which can be used as a suggested entry to replace. The second result can be a hit (1) 112 where the lookup value is in the CAM 64, and the entry number is an entry which has matched.

The LRU Logic 104 maintains a time-ordered list of the entry usage for the CAM 64. When an entry is loaded or matches with a lookup 108, it is marked as a MRU (Most Recently Used). A lookup that misses does not modify the LRU list.

Referring to FIG. 5, an exemplary CAM process 120 is shown. The programming engine 16a, for example, utilizes a 16-entry cache or CAM 64 with a LRU replacement policy to store a list of recently used variables working on eight (8) active contexts or threads at a time. The threads are executed in order, using a read phase 122 and a modify-write phase (not shown). During the read phase 122, a context requests a variable and the CAM 64 is checked to see if the needed variable is cached (124). If the CAM 64 indicates a hit, no read is necessary and the content of the CAM 64 gives the location of the variable in the CAM (126). Moreover, the updated value of the variable will be stored in the cache when this context becomes active for its modify-write phase, with the context reading the value directly from the CAM 64 (128).

On the other hand, if the CAM 64 indicates a miss during the read phase 122, a read of the needed variable is initiated (130).

Consequently, the execution time of the remaining seven (7) contexts being used to completely hide the latency of the read (132). Moreover, the variable is available at the modify write stage of this context (132). The write latency of the critical section is avoided since the variable is already valid in the CAM 64 if recently used. Next, the CAM is written or updated (134). The content of the CAM 64 provides the location of the LRU cached variable, with the new variable overriding the previously used variable (136).

Next, the context reads the value directly from the CAM 64 (136), and the context returns to the beginning of the read phase (122). Consequently, each programming engine becomes a pipeline stage, performing a specialized task of the packet processing, also monitoring the context or variable(s) used for particular tasks.

Context Pipelining

Referring to FIG. 6, a context pipeline 130 flow illustrates programming engines 16a-16h assigned to specific portions of a processing task of a packet or cell. The context for a specific assigned task is maintained on the programming engines 16a-16h using the CAM 64a-64c. The packets are processed in a pipelined fashion similar to an assembly line using the NN registers 54a-

54c to pass data from one programming engine to a subsequent, adjacent programming engine. Data is passed from one stage 132a to a subsequent stage 132b and then from stage 132b to stage 132c of the pipeline, and so forth. In other words, data is passed to the next stage of the pipeline allowing the steps in the processor cycle to overlap. In particular, while one instruction is being executed, the next instruction can be fetched, which means that more than one instruction can be in the "pipe" at any one time, each at a different stage of being processed.

For example, data can be passed forward from one programming engine 16 to the next programming engine 16 in the pipeline using the NN registers 54a-54c. This method of implementing pipelined processing has the advantage that the information included in the CAM 64a-64c for each stage 132a-c is consistently valid for all eight contexts of the pipeline stage. The context pipeline method may be utilized when minimal data from the packet being processed must advance through the context pipeline 130.

Other Embodiments

It is to be understood that while the example above has been described in conjunction with the detailed description thereof, the foregoing description is intended to illustrate and not limit the scope of the invention, which is defined by the scope of the

appended claims. Other aspects, advantages, and modifications are within the scope of the following claims.